Cから体感する78K0Rマイコン

(C)2010-2011 てきーらサンドム

R1.00 2010/5/18
R1.50 2011/2/27
R2.00 2011/3/29

4章　リアルなC言語（実用的プログラム構造）
4.1　プログラムの基本構造
4.2　基本構造の役割
4.3　基本構造の種類
4.3.1　巡回（ラウンドロビン）方式の基本
4.3.2　巡回＋モード番号方式
4.3.3　巡回＋関数リスト方式
4.3.4　巡回＋関数ポインタ方式
4.4　基本構造の設計例
4.4.1　基本構造の設計方法
4.4.2　基本構造の仕様例
4.4.3　基本構造のソース例
4.5　タスクの設計例
4.5.1　タスクの状態制御
4.5.2　状態制御用の関数
4.5.3　状態制御関数の使用方法
4.5.4　BTIMER_INTのタスク化
4.5.5　単純なメッセージ表示（Tmain4-1.c）の例
4.5.6　コラーツ予想（Tmain4-2.c）の例
4.5.7　COM通信（Tmain4-3.c）の例

４章　リアルなＣ言語

この章では実用的プログラム構造について説明します。

4.1　プログラムの基本構造

マイコンにも色々種類がありますが，特に８～１６ビット・マイクロコントローラではメモリ容量の制限や品種の多さから，標準的に搭載されるような OS（基本ソフト）がありません。

一応μITRON を始めとして何種類かの既製 OS はありますが，８～１６ビット・マイクロコントローラではそのような既製 OS を使わない場合が多くあります。これは OS を搭載する必要が無いという意味ではなく，メモリの制約が大きいため自分で OS に相当する機能を作り込まねばならないということです。

OS というと身構えてしまうかもしれませんが，結局やりたいことは下図のように，キー入力や表示など，色々な処理を平行して実行したいということで，OS はそのような平行処理を楽に実現できる環境であると言えます。

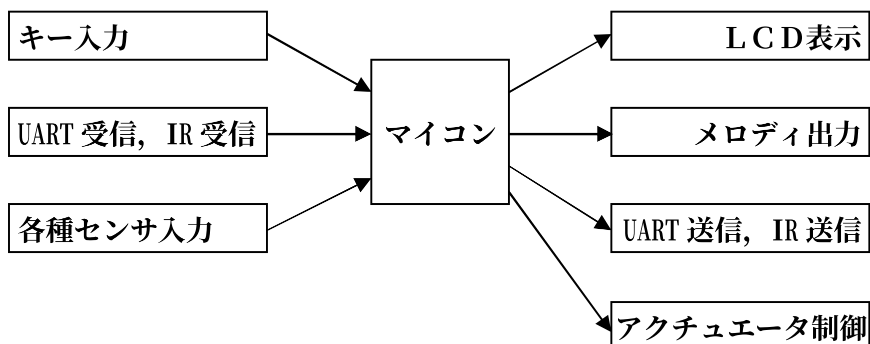


図 4－1．平行して動いて欲しい機能（１つが動作すると他が止まるのでは困る）

本書では，このように色々な処理を平行して動かす部分（OS に相当する部分）を「**基本構造**」と呼んでいます。OS と呼んでも良いのですが，後述の例だとわずか 150 バイト程度のプログラムであり，既製 OS の機能とは格段の差があるため控えめの用語にしました。

一方で，キー入力やＬＣＤ表示などの個々の処理は既製 OS と同じ扱いとして「**タスク**」と呼んでいます。OS が何であれ，仕様が同じなら実現したい機能も同じはずです。

4.2　基本構造の役割

色々な処理を平行して実行すると言っても，ハードウエアは１つしかないので，実際には下図のようにそれぞの処理を細切れにして実行することになります。

動作例：	■：動作中，□：待ち
キー入力	■□□□□□■□□□□□■□□□□□■□□□□□■□□□□□
ＬＣＤ表示	□■■□■□□□■□■■□■□■□■□□■■■□□■■□■□
メロディ演奏	□□□■□■□■□■□□□□■□■□□■□□□■□□□■□■

このように細切れにした処理の実行を制御するのが基本構造の役割です。
基本構造を作る上で知っておくと良い用語についてまず概説します。
ただし簡単な基本構造であれば，知らなくても済むので，後から読んでも良いです。

●タスク

平行して実行したいそれぞれの処理を言います。上の例では，キー入力やＬＣＤ表示にあたります。

●コンテキスト

細切れにされたタスクを再開するための情報・環境です。どこまで実行していたかとか，処理途中のデータなどを指します。

●プリエンプション

実行中のタスクより優先度が高いタスクが発生すると，実行中のタスクを強制的に中断して優先度の高いタスクを実行する方式を言います。

逆に実行中のタスクが実行権を開放し無い限り，他のタスクが実行できない方式をノン・プリエンプションと言います。

なお，割り込みによる処理はタスクとは言わずにハンドラと呼びます。

●スケジューラ，ディスパッチャ

いくつか実行可能なタスクがある場合に，どのタスクから実行するかを決めるのがスケジューラです。実際に実行タスクを切り替えるのがディスパッチャで，前に実行していたタスクのコンテキストを保存し，次に実行すべきタスクのコンテキストを復活してタスクを呼び出します。

4.3　基本構造の種類

基本構造には，次のような種類があります。

（1）巡回（ラウンドロビン）スケジュール，ノンプリエンプション

あらかじめ決まった順序でタスクを呼び出す方式で，なおかつ，タスクが自分で実行権を開放し無い限り他のタスクが実行できない方式です。

この方式は実装が容易でメモリ消費量も少ないことから，実用的プログラムの代表的な構造となっています。

さらに大きく二つに分けると，実行権を開放する時に自動変数（スタック領域）を開放する方式と開放しない（そのタスク専用のスタック領域を持つ）方式に分けられます。しかしタスクごとに専用のスタック領域を持てるぐらいの RAM があるのであれば，一般的には既製のリアルタイム OS を使う方が良いでしょう。特定のタスクだけ専用スタックを持ちたい場合は本方式の有効な手段として使えます。

本書ではスタックを開放する方式について，構造を詳しく解説しています。

（2）巡回スケジュール，プリエンプション

あらかじめ決まった順序でタスクを呼び出す方式ですが，タスクが自分で実行権を開放しない場合でも，所定時間経過したら強制的にコンテキストを保存して，次のタスクを実行する方式です。タイムシェアリングとかタイムスライスと呼ぶこともあります。

タスク専用のスタック領域が必要なので RAM の消費量は多めです。実際にこの方式専用の構造が作られることは少ないようですが，次に述べる方式の応用として実装されることは多いようです。

（3）イベント・ドリブン・プリエンプション

何かイベントが発生した時点でタスクの優先度判定を行い，優先度の低いタスクを中断して優先度の高いタスクを実行する方式です。リアルタイム OS の一般的方式です。

より具体的には，優先順位が高いタスクが生じるとか，優先順位が変化するようなイベントが発生すると，今まで実行していたタスクのコンテキスト（スタックやレジスタ内容）を保存し，優先するタスクのコンテキストを復活した上で優先するタスクを呼び出します。

巡回スケジュールへ応用する場合は，タイマ割り込みイベント内でタスクの実行順序を変更します。

この方式にするならば既製のリアルタイム OS を使った方が良いので，本書では説明を割愛します。

4.3.1　巡回（ラウンドロビン）方式の基本

各タスクを巡回的に呼び出す方式をラウンドロビン方式と言います。

以下は，キー入力（key），ＬＣＤ表示（LCD），メロディ演奏（Melody）の場合の基本的な例です。

```
int main(void) {
  while(1) {
    key();
    Melody();
    LCD();
  }
  return 0;
}
```

```
void LCD(void) {
    if (条件1) 処理A;
    else if (条件2) 処理B;
    else if (条件3) 処理C;
    else 処理D;
    return;
}
```

ラウンドロビン方式は，タスクの呼び出し順序をいちいち判定しないのでオーバーヘッド（無駄な時間）が少なく，へたに時間をかけて判定するより総合的な処理効率が良いことがしばしばあります。

コンテキスト（各タスクの実行継続に必要な情報）は各関数ごとに保存する必要があります。関数内の自動変数は１回呼び出されるごとに初期化されるので，例えば上記の処理AからBへデータを引き継ぎたければ，静的変数にデータを格納する必要があります。

リアルタイム応答性については巡回時間で決まります。例えばkey，LCD，Melodyの合計実行時間が1ms（0.001秒）以下であれば，応答速度は1ms以内ということになります。

巡回時間よりも速い応答が必要であれば，割り込みを使います。逆に言えば，最低限必要な高速応答性を最初から割り込みに任せると考えれば，巡回時間は遅くても良いということになります。

特に人間からみてリアルタイムに応答すればよい機器では，巡回時間は10～20msで十分です。巡回時間は10ms程度にして，COM通信など高速応答は割り込みで行うというのが，ロー・クラスのマイコンの一般的手法にもなっています。

タスクは，割り当てられた時間内に収まるように１回の処理を設計します。例えばLCD表示に割り当て可能な時間が1msであり，実際表示に必要な処理時間が4msであれば，少なくとも４分割以上に処理を分割する必要があります。

分割数は当然ながらマイコンの性能に依存します。78K0Rマイコンを20MHz動作させる場合は分割が必要になることは少ないでしょうが，コイン電池による長期間動作のために32kHz動作させるような場合には相応の分割が必要になるでしょう。

4.3.2　巡回＋モード番号方式

前述の例では if 文で処理分割を行っていますが，モード番号（条件番号）を定めて，それを switch 文で判定することもできます。そのタスクの例を次に示します。

```
static char mode;            /* モード番号保存用 */
void LCD(void) {
    switch(mode) {
      case LCD_A:   処理 A;
                    mode = 次の MODE 番号;
                    break;
      case LCD_B:   処理 B;
                    mode = 次の MODE 番号;
                    break;
      default:      処理 C;
                    mode = 次の MODE 番号;
                    break;
    }
    return;
}
```

この例では，次に実行すべきモード番号を固定的に代入していますが，条件によって変えることもできます。これにより複雑な状態遷移を比較的見通しよく記述することが出来ます。

モード番号は，#define 文によって定義するのが一般的です。モード数が少なければ，定義値はとびとびでも構いませんが，多い場合は 0 からの連番の方が switch 文のコンパイル効率が高くなると期待できます（特に最適化オプションで相対分岐を指定した場合）。モード番号を#define で定義しておけば，プログラム修正で番号が飛び飛びになっても付け直すことが容易です。

（例）

```
#define LCD_A       0       /* ストリームから文字取り出し */
#define LCD_B       1       /* 漢字コードのサーチ */
#define LCD_C       2       /* フォント転送 */
```

4.3.3　巡回＋関数リスト方式

これは，前述のモード番号方式の各処理(case 文ごとの処理)を別々の関数にし，その関数ポインタのリストから呼び出す関数を決める方式です。

呼び出し側の記述例を次に示します。

配列 funct は各タスクで定義された関数リストへのポインタです。

配列 mode にはタスクごとのモード番号を格納します。

```
char (**const funct[3])(void) = {key_list, LCD_list, Melody_list};
static char mode[3];
int main(void) {
    int i;
    while(1) for (i = 0; i < 3; i++) mode[i] = funct[i][mode[i]]();
    return 0;
}
```

各タスクは，次のようにモードごとに関数を分け，関数リスト（下の例では LCD_list）を作ります。

```
char (*const LCD_list[5])(void) = { LCD_A, LCD_B, LCD_C, LCD_D, LCD_E};
char LCD_A(void) {
    処理 A;
    return 次のモード番号;
}

・・・

char LCD_E(void) {
    処理 E;
    return 次のモード番号;
}
```

一般的にモード番号が 100 を超えることは希なので，１タスクあたり１バイトの RAM で状態を管理できます。

この方式は１つの関数サイズがコンパクトになるので，メンテナンスしやすくなります。

4.3.4　巡回＋関数ポインタ方式

前述の関数リスト方式は，モード番号と関数リストの並びをきちんと対応づける必要があります。モード番号の代わりに関数へのポインタで直接管理すれば，対比の手間が省けます。

ただし管理用のRAMが1タスクあたり2バイト（メモリ空間64Kバイト以内時）あるいは3～4バイト（メモリ空間64Kバイト超時）必要なので，RAMが非常に少ないマイコンでは実装が困難になります。

呼び出し側の例を次に示します。配列 funct が各タスクの実行関数へのポインタを保持します。初期値として最初に実行する関数を記述します。

```
void (*funct[3])(int) = {key_A, LCD_A, Melody_A};
int main(void) {
    int i;
    while(1) for (i = 0; i < 3; i++) funct[i](i);
    return 0;
}
```

タスク側ではモード（実行する関数）が変わる場合は配列 funct に次に実行する関数へのポインタを格納します。

```
extern void (*funct[])(int);
void LCD_A(int i) {
    処理A;
    funct[i] = 次に実行する関数名;
    return;
}

・・・

void LCD_E(int i) {
    処理E;
    funct[i] = 次に実行する関数名;
    return;
}
```

4.4　基本構造の設計例

実際の基本構造は，4.3.1～4.3.4の構造を組み合せたり，必要に応じてタイミング管理機能や資源管理機能を付け加えたりして作ります。

4.4.1　基本構造の設計方法

ここでは，巡回方式（ノン・プリエンプション）の基本構造の設計方法について説明します。

（1）最大巡回周期の決定

必要な応答性能から最大巡回周期を決定します。総タスク数N，個々のタスクの最大処理時間がt_i（ｉはタスクID）ならば，最大巡回時間Ｔは，

$$T = \Sigma^{N}_{i=0}(t_i)$$

従って，必要な応答性能からＴを決め，その範囲に収まるようにt_iを割り当てます。

なお，高速応答を割り込みに任せる場合は，最大巡回周期は10㎳程度で良いでしょう。

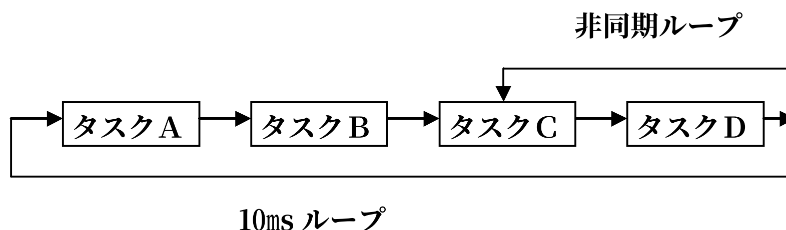
低消費電力化のためにクロック周波数を落す場合は，20㎳程度でも構わないでしょう。

このような値にするのは，キーのチャタリング除去，ブザーの発音時間制御などの単位として適当だからです。

（2）定周期タスクの有無決定

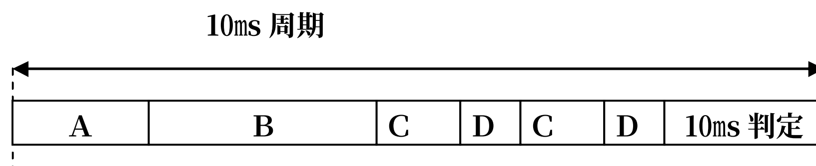
一定周期の処理をタイマ割り込み内で行うか，タスク内で行うかを決めます。厳密な周期で処理が必要であれば割り込み内で処理します。ただし複数の割り込みがある場合は，１つ１つは出来るだけ短時間で済ませる必要があり，割り込みで最低限の処理をした上で，その後の処理をタスクで行うという方法も考える必要があります。

また，キーのチャタリング除去のように，大体10㎳ぐらいで処理すればよいとか，多少の遅れは問題ならない処理は最初からタスクで処理してしまう方が良いです。このような場合は，タスクの巡回ループを10㎳周期ループと非同期ループに分けると良いでしょう。

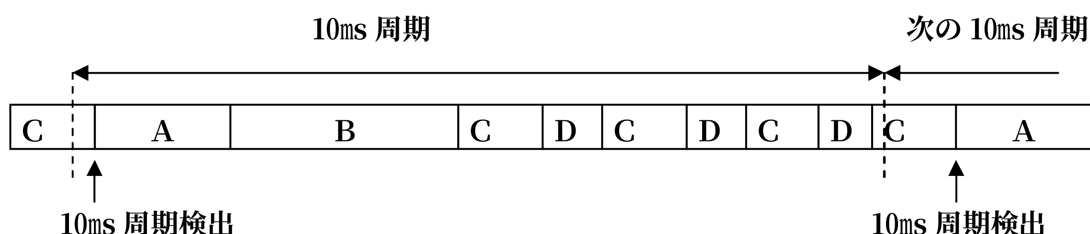


このような二重ループにする場合，タスクAの開始を 10ms に極力合わせるのか，多少遅れても許容するのかでタイミング管理の難易度が違ってきます。

タスクAの開始を 10ms に極力合わせる場合は，タスクC，Dは 10m 周期に対する残存時間をタイマから読み取って実行するかしないかを決定し，10ms 周期の最後では必ず 10ms 判定待ちだけをする期間を設けます。



一方，多少の遅れが許容できる場合は，非同期ループのタスクの処理時間を許容時間以内とすることで，タスク終了時だけの 10ms 判定で済みます。



なお，78K0R は１６ビット・タイマを豊富に持っているので上記のように最初から周期を 10ms と決めてしまっても問題になることは無いでしょう。例えば８ビット・タイマが１本しかないようなマイコンでは，クロック周波数の選定や異なるタイミング周期を要求する各タスクの巡回方法を決めるために苦労することになります。

（３）資源管理，タスク間通信機能

リアルタイム OS では資源管理やタスク間通信の機能が OS の機能として提供されています。しかしここで述べるノン・プリエンプション方式の場合は他のタスクによる割り込みが起きないため，資源管理やタスク間通信のための変数として一般的なグローバル変数や FILE（ストリーム）を使ってもほとんど問題は起きません。

ただしストリームを使う場合，書き込み側と読み出し側の速度が異なると破綻するので注意が必要です。3 章まではストリームが満杯になったら main 処理を止めておく（待ち状態にする）ということで暗黙的に回避しましたが，スタック開放型の巡回構造の場合はライブラリ関数内でタスクを止めるということが出来ません。

そこで，printf などのストリームに出力する関数を使う前には，必ずストリームの空き容量を確認するようにします。あるいは putc 関数のように１バイト単位でストリームに書き出して，エラーがあれば書き出しデータを静的変数に保管した上でタスクを終了します。

4.4.2　基本構造の仕様例

ここでは 3 章まで使用したドライバ・キットをなるべく流用する前提で基本構造を設計します。

4.3.4 項で説明した関数ポインタ方式をベースとして，いくつかのタイミング制御や状態制御を追加します。

基本的な考え方を次のようにします。

（1）巡回周期

最大 10ms とします。ただし一時的な 1～2ms の遅れを認めるものとします。

（2）巡回方法

10ms 周期と非同期の二重ループとします。ただし，タスクをどちらにでも動的に配置できるように，10ms 周期処理用のフラグを導入して次のように巡回します。

・10ms 周期の先頭で，10ms 周期のタスクの呼び出しを有効化します。

・巡回順序は固定とし，優先度の高い順に巡回するものとします。

・巡回の途中で 10ms 周期がきたら最初から巡回をやり直します。
　従って 10ms 中に 1 回は実行したいタスクは巡回の先頭に並べておきます。

・１巡しても 10ms に達しない場合は再巡回します。ただし実行すべきタスクがない場合は HALT 関数を呼び出して消費電力を下げます。

（3）タスクの状態制御

タスクは常に呼び出すのではなく，実行可能状態のタスクのみ呼び出します。

このためにタスクの状態管理フラグを導入して，次の４つのタスク状態に応じた呼び出し制御を行います。

・未登録状態
　実行するタスク（関数）へのポインタが登録されてない状態です。従って呼び出すことができません。

・実行可能状態
　巡回中にこの状態のタスクを呼び出します。

・次の 10ms 周期待ち状態
　この状態のタスクを 10ms 周期先頭で自動的に実行可能状態に変更します。
　巡回中にこの状態のタスクを見つけても呼び出しません。

・他タスクからの起床待ち状態
　この状態のタスクは呼び出しません。

4.4.3　基本構造のソース例

main 関数の記述例（main.c および TASK.h）を以下に示します。ドライバ・キットの TSOS フォルダに同等の記述があります。ここでは解説のため TASK.h の内容を main.c に含めてあります。

（1）タスク制御情報のテーブル（TASK_tcb）の定義

１つのタスクの情報を次の２つから構成しています。

①タスクへのポインタ（３バイト）

②状態制御フラグ１バイト

ただしタスクへのポインタ（関数へのポインタ）を変数定義で使うと４バイトの領域が確保されてしまうので，次のように実体とキャスト用を分けて宣言しています。

```
typedef struct {
  unsigned char addr[3];        /* タスクへのポインタ(実体) */
  unsigned char flags;          /* 状態制御フラグ */
} TASK_TCB;                     /* タスク制御情報の型を宣言 */

typedef void __far (*TASK_ADDR)(void);  /* タスクへのポインタを
                                           キャスト用として宣言 */

#definde TASK_NUMBER 3          /* タスク総数 */
TASK_TCB TASK_tcb[TASK_NUMBER]; /* タスク制御情報のテーブルを定義 */
```

CC78K0R においては，関数ポインタの領域が 4 バイト取られても実体の格納は 3 バイトだけなので，次のように代入しても状態制御フラグがつぶれることはありません。

```
((TASK_ADDR *)TASK_tcb)[n] = Tmain;          /* Tmain は関数名の例 */
```

このようなコンパイラ依存の記述は，本当は好ましくありませんが，メモリの少ないマイコンではやむを得ないこともあります。

状態制御フラグは，次の４種類のタスク状態を保持します。

```
#define TASK_ACTIVE    (0x01)       /* 実行可能 */
#define TASK_DELAY     (0x02)       /* 次の 10ms 周期待ち */
#define TASK_WAIT      (0x04)       /* 他タスクからの起床待ち */
#define TASK_EXIT      (0x00)       /* 未登録 */
```

（2）実行中タスクの制御情報へのポインタ（TASK_pt）の定義

タスク側において，実行状態や次に実行する関数を変更する場合に使用するポインタです。4.3.4 項の例では引数としてタスクへ渡していたものをグローバル変数で渡しています。

```
TASK_TCB *TASK_pt;
```

実際の記述では，TASK_tcb の直前に配置してあります。デバッガ動作中にプログラムを停止した場合，TASK_pt の内容を見て実行中のタスクの情報を得ることができます。

（3）10ms 周期開始フラグ（TASK_btf）の定義

10ms 定周期割り込みで 1 を立てるようにします。

```
_Bool    TASK_btf;
```

（4）main 関数（メイン・タスクの起動）

main 関数の先頭において，最初に実行するタスク（Tmain 関数）の起動設定をします。TASK.h に記述してあるタスクの登録マクロ（sta_tsk）を使って設定します。

```
void main (void) {
    TASK_TCB *p;          /* ワーク変数 */

    /* メイン・タスクが起動してなければ起動する */
    if (!ref_tsk(Tmain_ID))  sta_tsk(Tmain_ID, Tmain, TASK_ACTIVE);
```

“メイン・タスクが起動してなければ”というのは，タイム・オーバーでタスクの強制打ち切りが発生した時に main 関数から再実行してもメイン・タスクに影響が出ないようにするための措置です。

本当は TASK_tcb の初期化データとして((long)Tmain |TASK_ACTIVE << 24)が記述できれば起動設定は省けたのですが，CC78K0R(W2.12)ではエラーが出てしまいます。しかしプログラム中にこの記述をしてもエラーはでませんので，静的変数の初期化において何か特別の扱いをしているようです（Tmain だけなら格納可能で，そのアセンブラ・ソースを見ると特殊な DG 疑似命令によって領域が確保されています）。

（５）main 関数（10ms 周期処理）

スケジューラのメイン・ループの最初で，ウオッチドッグ・タイマをクリアします。
その次に 10ms 周期フラグが立っている場合は，10ms 周期待ちをしているタスクを起床します。

```
while(1) {
    main_loop:
    WDTE = 0xac;             /* ウオッチドッグ・タイマのクリア */
    if (TASK_btf)   {        /* 10ms 周期処理判定 */
        TASK_btf = 0;
        for (p = &TASK_tcb[0]; p < &TASK_tcb[TASK_NUMBER]; p++) {
            if (p->flags == TASK_DELAY) p->flags = TASK_ACTIVE;
        }
    }
```

for ループの部分は，int i を定義して(i=0; i<TASK_NUMBER; i++)と書いても良いのですが，オーバーヘッドを極力少なくするためにポインタを使っています。

（６）main 関数（スケジューラ）

実行可能状態になっているタスクを順に呼び出します。
ただし 10ms 周期フラグが立っている場合は，メイン・ループの最初に戻ります。

```
    for (p = &TASK_tcb[0]; p < &TASK_tcb[TASK_NUMBER]; p++) {
      if (TASK_btf) goto main_loop;
      if (p->flags == TASK_ACTIVE) {
            TASK_pt = p;
            (*(TASK_ADDR *)p)();
      }
    }
```

ここで，TASK_pt を引数ではなく静的変数でタスクへ渡すようにしたのは，オーバーヘッド削減の他に，デバッガ停止時にどのタスクを実行中だったか判別しやすいという理由があるためです。

（７）main 関数（HALT 判定）

タスク呼び出しが一巡後，まだ実行可能なタスクがあれば再巡回します。実行可能なタスクが１つもなければ，消費電力を低減するために HALT 関数を呼び出します。

実行可能なタスクの有無を調べるときは，割り込みにより調査済のタスクの状態が変わらないように割り込み禁止にしておきます。実行可能タスクがなければ割り込み禁止のまま HALT 関数を呼びます。

78K0R は割り込み禁止中に HALT しても，割り込みマスク許可状態の割り込みがすでに入って保留になっていればすぐに HALT を抜けます。もちろん HALT した後で割り込みマスク許可状態の割り込みが入ってもすぐ抜けます。

```
		TASK_wk = 0;
		DI();
		for (p = &TASK_tcb[0]; p < &TASK_tcb[TASK_NUMBER]; p++) {
			TASK_wk |= p->flags;
		}
		if (!(TASK_wk & TASK_ACTIVE) && !TASK_btf) {
			HALT();
		}
		EI();
	}
	return;
}
```

（８）諸元

ドライバ・キット\TSOS\main.c のコンパイル結果をまとめておきます。

ROM サイズ：	150［バイト］	
RAM サイズ：	通常エリア	２＋タスク数ｘ４［バイト］
	SADDRP 空間	２［バイト］
	ビット空間	１［ビット］
巡回速度　：	53＋タスク数ｘ65［クロック］（タスク側オーバーヘッド含まず）	
タスク配置：	__far 空間も対応	

この程度のオーバーヘッドならそこそこ広い用途で使えると思います。

4.5　タスクの設計例

基本構造の仕様に合わせてタスクの構造を決めます。ここでは 4.4.2 項の基本構造に対応したタスクの設計例について説明します。

4.5.1　タスクの状態制御

タスクの状態は，タスク制御情報テーブルのフラグ（TASK_tcb[].flags）で管理します。状態には次の４種類があります。

・未登録（終了）状態
　タスクへのポインタを登録してない状態です。従って呼び出されません。
・実行可能状態（実行中含む）
　巡回中に呼び出され，実行状態になります。
・次の 10ms 周期待ち状態
　巡回中に呼び出されません。次の 10ms 周期先頭で自動的に実行可能状態になります。
・他タスクからの起床待ち状態
　巡回中に呼び出されません。他タスクから実行可能状態に変更されるのを待ちます。

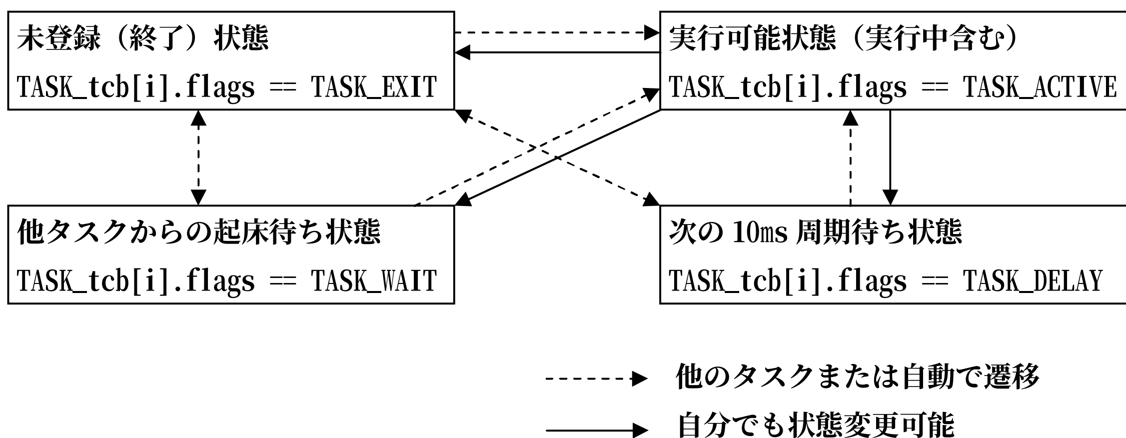


図 4.5.1-1　タスクの状態遷移

タスク制御情報テーブル（TASK_tcb[]）の何番目にどのタスクを登録するかは，TASK.h で番号（タスク ID）を定義します。

タスク ID は，0 以上，タスク制御情報テーブルのサイズ未満で付与します。優先度の高いタスクほど若い番号とし，特に 10ms 中に必ず 1 回は実行したいタスクは 0 から順に並べるようにします。

なお，実行中のタスクが自分の状態を変更する場合，いちいち自分のタスク ID を記述するのは面倒なので，TASK_pt（実行中のタスク制御情報テーブルへのポインタ）で指定できるようにしています。

4.5.2　状態制御用の関数

タスクの中で，タスク状態 TASK_tcb[].flags を直接変更するのは好ましい記述ではありません。そこで状態制御用の関数を定義して使用することにします。実際には簡単のため #define マクロ関数を定義しています。

関数の名前は，あえてμITRON 風にしました。今の時代にこのような短縮形の名前が良いかどうか分かりませんが，将来μITRON を使うかもしれないので，なるべく違和感の無い名前にしました。

表 4.5.2 -1　状態制御用の関数一覧

区分	関数名（引数）	内容
他タスクの操作	sta_tsk(タスク ID, 実行関数名，状態名)	タスクを登録します。登録済みのタスクを強制的に変更することも考えて，登録済みかどうかの判定はしていません。（状態名は図 4.5.1 -1 の TASK_xxx を使用）。
	sus_tsk(タスク ID)	指定タスクを起床待ちに変更します。
	wup_tsk(タスク ID), iwup_tsk(タスク ID)	指定タスクを起床します。iwup_tsk は割り込み内で使用します。（i 無し関数名は割り込み内での使用は禁止）
	ter_tsk(タスク ID)	指定タスクの登録を強制的に削除します。指定タスクで使用していたリソースが開放されない可能性があるので，本関数を使わずに sta_tsk でリソース開放関数へ強制遷移させるのが望ましいです。
	ref_tsk(タスク ID)	指定タスクの状態を取得します。
自タスクの操作	dly_tsk()	次の 10ms 周期待ちにします。
	slp_tsk()	他タスクからの起床待ちにします。
	ext_tsk()	タスクを終了します（登録を削除します）。
	上記変更無しで return	実行可能状態を継続します。すなわち再度巡回して呼び出してもらうことを要求します。
	trs_tsk(実行関数名)	次の巡回で呼び出してもらう関数名を指定します。

なお，状態変更の全ての組み合わせを関数にしていません。例えば，他のタスクを“次の 10ms 周期待ち”にしたければ，上記マクロ記述を参照して適当な関数を作るようにします。

4.5.3　状態制御関数の使用方法

（１）タスクの起動

一番最初は，Tmain 関数のみ実行可能状態になっています。この関数から他のタスクを sta_tsk 関数で登録して起動します。

（２）自タスクの通常の操作

通常は，return の直前に状態変更を行います。

①次の巡回で実行する関数を変更したい場合は，trs_tsk 関数で指定します。
指定しない場合は，現在と同じ関数が呼ばれます。

②すぐに呼び出して欲しい場合以外は，次の関数で状態変更をします。
dly_tsk（10ms 待ち），slp_tsk（他タスクからの起床待ち），ext_tsk（終了）
これらを指定しない場合は，すぐ再巡回して呼び出されます。ただし“すぐ”と言ってもタスクが多ければ最低でも数十μs 以上かかり，他にもすぐ呼び出して欲しい優先度の高いタスクがあればその実行時間もかかります。

③最後は，return;　を記述します。

なお，dly_tsk を良く使う場合，#define DLY_tsk { dly_tsk(); return; }　を定義しても構いませんが，次の構文では注意します。

```
if (条件) DLY_tsk();     →  {DLY_tsk();}と書かないとエラーになる。
else      return;
```

（３）割り込みで起床する場合

割り込みで起床されたタスクが待ち状態で終了するような場合は，次のように最初に状態変更しないと，処理途中に再度起床要求があっても受け付けられなくなります。

```
void funct(void) {
    slp_tsk();
    割り込みに連動した処理；
    return;
}
```

（４）再巡回を待つのが非効率な場合

タスク数が多くなると再巡回まで数十μs 以上かかります。このオーバーヘッドを無くすためになるべくタスク内でループしたい場合は，次のように 10ms 周期フラグを判定するようにします。

```
while (ループ条件 && !TASK_btf) 処理;
```

4.5.4　BTIMER_INT のタスク化

3章までは各種ドライバ関数を10ms周期割り込み関数BTIMER_INTの中に詰め込んでいました。この構造が一概に悪いとは言いませんが，ドライバ関数とmain関数の間で干渉を起こし易く（リエントラント性やポートのビット操作干渉），何かと神経を使います。

そこでBTIMER_INTからドライバを切り離して，ドライバをタスク化します。

（1）BTIMER_INTの変更

BTIMER_INTで必須なのはプロセッサ時間のカウントのみです。それ以降に従来記述していたドライバ関数の呼び出しは切離します。

ただし，リビジョンR2.00では次の記述も加えてあります。

①ダイナミック・スキャン用の記述（将来の拡張用）

多桁の数値LEDを表示する用途ではダイナミック・スキャン手法を用います。

この場合，10msをさらに何分周かした時間でタイマ割り込みをかけて，スキャン用の関数を呼び出します。

10ms周期に到達したら従来の処理を行います。

②タスクのオーバーラン検出と打ち切り

4.4節で示した基本構造は，タスクが自らreturnしない限り他のタスクが実行できません。すなわち，１箇所でも無限ループに陥れば全体が停止するという，いわゆるフリーズとかハングアップという現象が出ます。

この現象を緩和するために，10ms待っても再巡回が発生しない場合は実行中のタスクを打ち切ります。打ち切り部分をアセンブラで記述してあるためワーニングが出ますが問題はありません。

（2）定周期処理タスク（Ttimer.c）

本来はドライバの種類ごとに別タスクに分けた方が適切なタイミング管理が出来るのですが，本版では簡単のためLCDとそれ以外の定周期処理の２つに分けます。

定周期処理タスクは次の２つの関数で構成しています。

①Ttimer（初期化）

基本周期タイマおよびRTC（時計機能）をオープンします。

②Ttimer_exe（10msに１回の処理）

ドライバ・リスト(driver_list.h)に応じて次の４つを組み込んでいます。

- キー入力およびコード変換
- メロディ演奏制御
- UART（COM通信）制御
- IR（赤外通信）制御

（３）LCD 表示タスク（Tlcd.c）

次の２つの関数で構成しています。

①Tlcd（初期化）

LCD オープン関数を呼び出します。

LCD ドライバはタイマが動いてないとハングアップするので，タイマの動作確認も行っています。

②Tlcd_exe（表示）

これも最初にタイマの動作確認を行っています。タイマが停止していたら本タスクも終了するようにしています。

キーエコー表示は 10ms 中に１文字あるかないか程度のため，データがあれば無条件で実行します。

通常の表示（stdout ストリームの表示）は量が多い場合があるので，10ms 周期フラグを監視しながら処理を行っています。

4.5.5　単純なメッセージ表示（Tmain4-1.c）の例

従来 main 関数で記述していた内容は Tmain 関数で記述します。ここでは，main2-2.c をタスク化した例を示します。Tmain，Tmain_exe の２つの関数で構成しています。

（１）Tmain

ドライバ・タスク Ttimer，Tlcd を登録（起動要求）しています。

なお，タスク ID は TASK.h 内で記述しています。

```
TASK_STATE __far Tmain(void) {
        sta_tsk(Ttimer_ID, Ttimer, TASK_ACTIVE);
        sta_tsk(Tlcd_ID, Tlcd, TASK_ACTIVE);

        trs_tsk(Tmain_exe);
        return;
}
```

ここではドライバ・タスクを Tmain で登録していますが，もちろん main 関数の最初で登録することもできます。その場合は次に示す Tmain_exe の内容を Tmain に書けば良いです。

（2）Tmain_exe

printf の出力ストリームである stdout に空きがあるかどうか調べて，空きがあれば printf を実行します。

```
TASK_STATE __far Tmain_exe(void) {

        if (FILE_space(stdout) < 50) return;     /* 空きチェック */

        printf( "今日も元気だ！\n"
                "全てが順調だ！\n"
                "A\b\"\t ってね。");


        ext_tsk();
        return;
}
```

FILE_space 関数は，ストリームの空きバイト数を返す関数です（R2.00 で追加）。

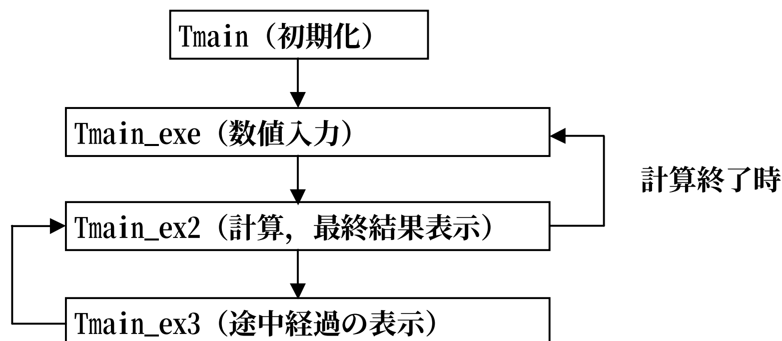
他に LCD 表示するタスクはありませんので，ここではチェックは必須ではありませんが，標準的には必要という意味で入れてあります。

従来，stdout が満杯なら空きが出来るまで待つようにしていましたが，タスク版では関数内での待ちは望ましくないので，待たないように変更しています。

最後は ext_tsk()で終わっているので，メイン・タスクはこれ以上は動作しません。しかしドライバは止めてないので，キーエコーの表示は引き続き行えます。

4.5.6　コラーツ予想（Tmain4-2.c）の例

main2-6.c に数値入力を追加した上でタスク化した例です。次の４つの関数構成になっています。



（1）Tmain（初期化）

Tmain4-1.c の Tmain 関数と同じです。Ttimer，Tlcd を起動します。

（2）Tmain_exe（数値入力）

計算の初期値を変数 Collatz に格納し，カウント変数 count をクリアします。

なお，数値入力を催促するプロンプト表示の制御のためにビット変数 prompt も定義しています。

fgets 関数は，従来は\n 入力完了（stdin の場合は ENT キー入力完了）まで待っていましたが，タスク版では\n が検出できなければ，いったん NULL を返すようにしています。

```
static int      Collatz, count;         /* 計算用の変数，カウント用変数 */
_Bool           prompt;                 /* プロンプト表示制御用 */

TASK_STATE __far Tmain_exe(void) {
        char wk[10];

        if (!prompt) {                  /* プロンプト表示済かどうか判定 */
                if (FILE_space(stdout) < 15) return;
                printf("Num=    \b\b\b\b");
        }
        prompt = 1;
        if (fgets(wk, sizeof wk, stdin) == NULL) {      /* キー入力判定 */
                dly_tsk();
                return;
        }
        prompt = 0;
        if ((Collatz = atoi(wk)) < 1) return;   /* 0 以下の入力はやり直し */
        count = 0;                              /* カウンタ・クリア */

        trs_tsk(Tmain_ex2);
        return;
}
```

（3）Tmain_ex2（計算，最終結果表示）

計算が終了したかどうか判定し，終了なら結果を表示します。キー入力した初期値が大きいと，途中でオーバーフローする（負になる）ことがあるので，その場合はエラー表示を行います。

計算が終了してなければ，計算を行った上で途中経過を表示する関数を呼び出します。

```
TASK_STATE __far Tmain_ex2(void) {
    if (Collatz <= 1) {                          /* 計算完了の判定*/
        if (FILE_space(stdout) < 10) return;
        if (Collatz == 1) printf("n=%d, ", count);      /* カウント結果表示*/
        else              printf("error, ");            /* エラー表示 */
        trs_tsk(Tmain_exe);                             /* キー入力に戻る */
        return;
    }

    Collatz = Collatz & 1 ?                      /* 計算 */
              Collatz * 3 + 1 :
              Collatz / 2;
    count++;                                     /* カウント */
    trs_tsk(Tmain_ex3);                          /* 途中経過表示 */
    return;
}
```

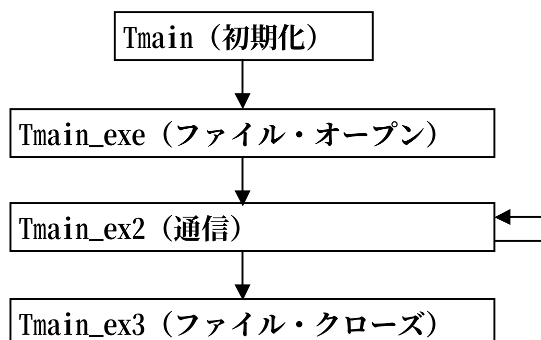
（4）Tmain_ex3（途中経過の表示）

stdout に空きがあるか確認の上で表示を行います。

```
TASK_STATE __far Tmain_ex3(void) {
        if (FILE_space(stdout) < 10) return;
        printf("%d,", Collatz);
        trs_tsk(Tmain_ex2);                      /* 計算に戻る */
        return;
}
```

4.5.7 COM 通信（Tmain4-3.c）の例

main3-10A.c をタスク化した例です。モード番号方式との併用例です。
次の 4 つの関数構成になっています。



（1）Tmain（初期化）

Tmain4-1.c の Tmain 関数と同じです。Ttimer，Tlcd を起動します。

（2）Tmain_exe（ファイル・オープン）

main3-10A.c のファイル・オープン部分とほぼ同じです。最初はストリームは空なので，ストリームの空きチェックは省略しています。

```
static FILE     *fpr, *fpw;
static char     tmp[60];
static char     mode;

TASK_STATE __far Tmain_exe(void) {
        fpw = fopen("COM:", "wb");
        fpr = fopen("COM:", "rb");
        if (fpr == NULL  || fpw == NULL) {
                printf("error\n");      /* ストリーム空きチェックは省略 */
                ext_tsk();
                return;
        }
        fputs("通信開始\r\n", fpw);     /* ストリーム空きチェックは省略 */
        trs_tsk(Tmain_ex2);
        return;
}
```

（３）Tmain_ex2（通信）

受信，LCD 表示，ループバックの３状態をモード管理しています。モード番号は手抜きをして#define 定義していません。

従来は，fgets 関数内で\n を受信するまで待っていましたが，タスク版ではストリームのデータが指定サイズ未満しかなく，かつ\n も検出できない場合は NULL を返すようにしています。

また，fputs 関数は，引き渡された文字列の長さ以上の空きがストリームになければ EOF を返すようにしています。

```
TASK_STATE __far Tmain_ex2(void) {
    switch (mode) {
        case 0:         /*** 受信 ***/
                if (fgets(tmp, sizeof(tmp), fpr) == NULL) return;
                mode = 1;
                return;
        case 1:         /*** LCD 表示 ***/
                if (FILE_space(stdout) < sizeof(tmp)) return;
                printf("%s", tmp);
                mode = 2;
                return;
        default:        /*** ループバック ***/
                if (fputs(tmp, fpw) == EOF) return;
                if (!strcmp(tmp, "end\r\n")) trs_tsk(Tmain_ex3);
                mode = 0;
                return;
    }
    return;
}
```

この例の場合，各 case 文内の return は break と書いても同じです。一般的には case 文内では break で抜けるように書きますが，ここではタスク制御を意識して return を記述しています。

（４）Tmain_ex3（ファイル・クローズ）

終了通知，クローズ，終了表示の３状態をモード管理しています。

従来は，UART 送信クローズは，クローズ関数内で送信終了するまで待っていましたが，タスク版ではいったん UART_CLOSE_RETRY を返すようにしました。

```
TASK_STATE __far Tmain_ex3(void) {
    switch (mode) {
        case 0:         /* 終了通知 */
                if (fputs("通信終了\r\n", fpw) == EOF) return;
                mode = 3;
                return;
        case 3:         /* クローズ */
                if (fclose(fpw) == UART_CLOSE_RETRY) return;
                fclose(fpr);
                mode = 4;
                return;
        default:        /* 終了表示 */
                if (FILE_space(stdout) < 10) return;
                printf("closed\n");
                ext_tsk();
                return;
    }
    return;
}
```